**In het eerste deel van deze serie (Java Magazine nr. 1, april 2006) lieten Jesse Houwing en Peter van den Berkmortel zien dat je met Reguliere Expressies veel code kunt besparen. We zagen onder andere hoe een RegEx wordt opgebouwd en waar deze uit bestaat. We bekeken Character Classes, Quantifiers en Boundaries. In dit tweede artikel zullen de auteurs ingaan op look-arounds, switches, Character Classes voor Unicode ondersteuning en het vervangen van tekst.**



# Regular Expressions in Java (2)

## *Herkenning van patronen*

In het vorige artikel zagen we dat een word boundary een match maakt tussen een woord en de witruimte tussen woorden, maar zonder dat er daadwerkelijk karakters gematcht worden. Een word boundary is een voorbeeld van look-around. Er zijn vier varianten van look-around. Look-ahead kijkt vooruit, look-behind kijkt terug. Beide kunnen bovendien zowel positief als negatief zijn. Look-arounds matchen geen karakters. In plaats daarvan matchen ze op een positie in de tekst. Een word-boundary bijvoorbeeld matcht op de positie waar een woord begint of eindigt (in RegEx termen: de overgang tussen een niet-woord karakter en een woord-karakter of vice versa).

**LOOK-AROUNDS** In een RegEx zien look-arounds er als volgt uit:

| | |
|---|---|
| (?=..) | Positive look-ahead |
| (?!..) | Negative look-ahead |
| (?<=..) | Positive look-behind |
| (?<!..) | Negative look-behind |

Hoe deze RegEx-elementen zich gedragen kan het beste met een voorbeeld worden duidelijk gemaakt.

```
// Voorbeeld Quantifiers
// Matches staan tussen kleiner en
//   groter dan tekens.
Input: Dit is een mooie zin, zeg!

Pattern: (?=\w).{3}
Matches: <Dit> <is > <een> <moo> <ie > <zin>
<zeg>

Pattern: (?!\w).{3}
Matches: < is> < ee> < mo> < zi> <, z>

Pattern: (?<=\w).{3}
Matches: <it > <s e> <en > <ooi> <e z> <in,>
<eg!>

Pattern: (?<!\w).{3}
Matches: <Dit> <is > <een> <moo> <zin> < ze>

// Postcode voorbeeld met
// negative look-ahead
Pattern: (?!0)\d{4}[ ]?[A-Z]{2}
Uitleg: het eerste karakter mag geen nul
zijn!
```

Bij het positive look-ahead voorbeeld zien we dat een match altijd begint met een woordkarakter. Look-ahead kijkt dus vooruit. Bij het negative look-ahead voorbeeld zien we dat een match juist begint met een niet woord karakter. Negative betekent dus dat er juist *geen* match gemaakt kan worden met de RegEx tussen de ronde haken. Bij het positive look-behind voorbeeld zien we dat een match altijd moet worden voorafgegaan door

een woord karakter. Look-behind kijkt dus terug. Bij het negative look-behind voorbeeld zien we dat een match juist moet worden voorafgegaan door een niet woord karakter.

Een word boundary kan nu als volgt gedefinieerd worden met look-arounds: `((?<=\w)(?!\w)| (?<!\w)(?=\w))`.

In het eerste deel `((?<=\w)(?!\w))` kijken we of er voor de huidige positie een woordkarakter staat (met look-behind) en na de huidige positie een niet woordkarakter (met look-ahead). In het tweede deel `((?<!\w)(?=\w))` is dit juist andersom.

**SWITCHES** Met switches kan het gedrag van een RegEx worden aangepast. Er zijn twee manieren om de switches aan of uit te zetten: als extra parameter bij het aanmaken van het Pattern (overloaded `compile` methode), of via een embedded flag in de RegEx zelf. De eerste methode is altijd globaal, dus geldig voor het hele Pattern. Een embedded flag kan zowel globaal zijn, als voor een specifiek subdeel van de RegEx. De volgende switches zijn beschikbaar:

CASE_INSENSITIVE ( ?i)
Case onafhankelijk matchen van karakters. Gaat uit van de US-ASCII charset.
UNICODE_CASE ( ?u)
Bij case onafhankelijk matchen wordt uitgegaan van de Unicode standaard, in plaats van US-ASCII.
COMMENTS ( ?x)
Witruimte en commentaar in de RegEx wordt genegeerd. Commentaar begint met een #-teken en eindigt bij het einde van een regel.
MULTILINE (?m)
Zorgt ervoor dat ^ en $ (ook) matchen op een regel einde (line terminator) en niet alleen op het begin dan wel einde van de input.
UNIX_LINES (?d)
Alleen de unix regel einde (\n) wordt herkent als regel einde (line terminator).
LITERAL <geen>
Het Pattern wordt als een letterlijke karakter reeks gezien. Speciale karakters hebben geen betekenis.
DOTALL (?s)
Default matcht een punt (.) alles behalve de regel einde (line terminator). In dotall mode wordt met een punt (zie Character Classes in het eerste artikel) alles gematcht (dus ook een regel einde).
In Perl heet deze mode de single-line mode.

```
// Voorbeeld Switches
Pattern pattern = Pattern.compile(
  "[1-9]\\d{3}[ ]?[A-Z]{2}",
  CASE_INSENSITIVE);

// Matches: postcodes met grote en
// kleine letters!

// Idem, maar nu embedded:
Pattern: (?i)[1-9]\d{3}[ ]?[A-Z]{2}

// Idem, maar niet globaal:
Pattern: [1-9]\d{3}[ ]?(?i:[A-Z]{2})
```

In de niet-globale versie van de switch, is een dubbele punt noodzakelijk om aan te geven waar de switch(es) eindigen en de RegEx begint. Switches kunnen ook uitgezet worden, maar alleen in de embedded variant. Er moet dan een minteken voor de switch gezet worden.

```
// Voorbeeld Switches uitzetten:
Pattern: (?i-u)[1-9]\d{3}[ ]?[A-Z]{2}

// Matches: postcodes met grote en
// kleine letters, maar er wordt
// uitgegaan van de US-ASCII char-
// set (en niet van Unicode).
```

**QUOTATION** Quotation wordt gebruikt om een reeks karakters automatisch te escapen. Speciale karakters zoals de accolade en de rechte blokhaak worden dan letterlijk genomen.

| | |
|---|---|
| \Q | Begin van quotation |
| \E | Einde van quotation |

```
// Voorbeeld Quotation:
Pattern: \w\Q{3}\E
Matches: A{3}, maar niet B{5}, AAA
```

**UNICODE** De volgende Character Classes zijn specifiek voor de ondersteuning van Unicode karakters:

\p{L} Een Unicode karakter.
\p{In<Blok>}
Een Unicode karakter in blok Blok. \p{InGreek} is dus een Griekse letter.
\P{In<Blok>}
Een Unicode karakter die niet in blok Blok

voorkomt.
\p{L<Categorie>}
Een Unicode karakter in categorie Categorie. \p{Lu} is dus een uppercase karakter.
\p{Sc} Currency symbolen ($, , ¥).
\p{Sm} Math symbolen (¬, ×)

De Unicode-blokken zijn terug te vinden in de Java-API (http://java.sun.com/j2se/ 1.5.0/docs/api/java/lang/Character.UnicodeBlock.html). De verschillende blokken zijn daar gedefinieerd als constanten. De meegegeven bloknaam wordt bepaald via de `forName` methode. De categorieën komen overeen met de java.lang. Character classes, die ook nog via hun eigen classes te benaderen zijn:

\p{java<Method>}
Unicode karakters conform java.lang.Character.isMethod() methode.

Voorbeelden van dergelijke methodes zijn: isLowerCase(), isUpperCase(), isLetter(). De categorie is telkens de derde letter van de methodenaam.

## Datum controle

Hoe kan RegEx nu het beste gebruikt worden om, bijvoorbeeld, een datum controle te doen? Je kunt eenvoudig het patroon (twee cijfers, scheidingsteken, twee cijfers, scheidingsteken en vier cijfers) controleren. Maar dat levert een controle op met een beperkte functionaliteit.

We willen een controle die de gehele datum valideert. Hoever kunnen we daarmee komen met RegEx? Laten we beginnen met de RegEx om het datum patroon te controleren. Dat ziet er als volgt uit: `\d{2}([-/])\d{2}\1\d{4}`.

We staan voor het scheidingsteken zowel een minteken als een schuine streep toe. Echter, het tweede scheidingsteken moet wel hetzelfde teken zijn als het eerste scheidingsteken! Dat dwingen we af door het eerste scheidingsteken in een capture-groep op te nemen en het tweede scheidingsteken via een back-reference te matchen.

Met dit patroon wordt ook 99-99-0000 geaccepteerd. Dat willen we natuurlijk niet. De dag dient maximaal 31 te zijn. De RegEx hiervoor ziet er zo uit: `(?!00)(?:[0-2]\d)|(?:3[01])`.

De look-ahead zorgt ervoor dat 00 geen geldige waarde is. Voor de rest zijn 01 t/m 29 geldige waarden. Of 30 en 31. De maand dient maximaal 12 te zijn. De RegEx hiervoor ziet er zo uit:

```
(?:0[1-9])|(?:1[012]).
```

Naast dag 00 zijn er nog andere verboden combinaties. Die kunnen we ook uitsluiten met negative look-ahead: `(?!31.(?:0[2469])|(?:11))(?!30.02).`

Aangezien we met negative look-ahead geen karakters matchen, kunnen we meerdere negative look-aheads achter elkaar plaatsen, zonder alternation te

> Bij een postive look-ahead zien we dat een match altijd begint met een woordkarakter

hoeven toepassen. Nu blijft alleen 29 februari over als mogelijke verboden combinatie. Maar die combinatie is alleen verboden als het geen schrikkeljaar is. Kunnen we met RegEx controleren of een jaar een schrikkeljaar is? Of liever nog, of het *geen* schrikkeljaar is? Ja, dat kan, met een opsomming van de verboden combinaties:

```
(?!29.02.(?:\d{3}[13579])|      (?:\d{2}(?!00)[02468][026])|      (?:\d{2}[13579][048])|   (?:[02468][026]00)|(?:[13579][048]00)).
```

Dit stukje RegEx is vooral lang, omdat RegEx zich niet zo goed leent om min of meer arbitraire data te matchen. Hoe ziet de totale RegEx er nu uit?

```
(?!00)
(?!31.(?:0[2469])|(?:11))
(?!30.02)
(?!29.02.
  (?:\d{3}[13579])|
  (?:\d{2}(?!00)[02468][026])|
  (?:\d{2}[13579][048])|
  (?:[02468][026]00)|
  (?:[13579][048]00)
)
(?:[0-2]\d)|(?:3[01])
([-/])
(?:0[1-9])|(?:1[012])
\1
\d{4}
```

Om de leesbaarheid te vergroten is extra witruimte toegevoegd. Deze moet natuurlijk weggelaten worden (of de COMMENTS switch moet worden aangezet). Het voorbeeld maakt duidelijk dat je niet alles in één keer kan en wil controleren. Maar met look-around hoeft dat ook niet. In plaats daarvan zoek je naar het meest spe-

cifieke patroon waar alles nog onder valt en benoem dan de afwijkingen in positieve of negatieve zin. Als dat lastig wordt, maak dan gebruik van alternation om de verschillende alternatieven op te noemen. In de controle op schrikkeljaar zien we dat we zaken toch nog redelijk kunnen samenvatten. Uiteindelijk hadden we slechts vijf verschillende alternatieven nodig om het schrikkeljaar te controleren.

**ZOEK EN VERVANG** Een laatste toepassing voor Reguliere Expressies is het zoeken en vervangen van patronen in tekst. De `String` klasse biedt enkel de mogelijkheid om een opgegeven set karakters te vervangen met een ander karakter en de mogelijkheid om een tekst te vervangen met een ander stuk tekst. Geavanceerde controles of het opgeven van een patroon dat vervangen moet worden, waren tot voor versie 1.4 van Java nog niet zonder meer mogelijk. Met de komst van Reguliere Expressies in Java is de `String` klasse uitgebreid met de mogelijkheid om tekst te vervangen op basis van reguliere expressies. De `String` klasse maakt daarbij gebruik van een `Matcher` om tekst te vervangen.

Over het algemeen is het beter om direct gebruik te maken van Pattern en Matcher, omdat je dan gebruik kan maken van een precompiled Pattern

Over het algemeen is het beter om direct gebruik te maken van Pattern en Matcher, omdat je dan gebruik kan maken van een precompiled Pattern. De performance daarvan is aanmerkelijk beter. Met het postcodevoorbeeld onder Java 5 is het verschil gemiddeld een factor 7! In principe zijn er vijf verschillende manieren om stukken tekst te manipuleren door middel van reguliere expressies:

1. Het zoeken van een vaste tekst en deze vervangen door een andere tekst.
2. Het zoeken van een patroon en dit vervangen door een andere tekst.
3. Het zoeken van een of meerdere (deel) patronen en deze invoegen in een template.
4. Het zoeken van een deelpatroon en dit verwijderen uit de tekst.
5. Het zoeken van een (deel)patroon en dit via een zelf geprogrammeerd algoritme vervangen door iets anders.

De eerste methode is vrijwel identiek aan de `String.replace()` methode.

```
// Vervang alle voorkomens van
// “foo” door “bar”:
String text = “foo test foo”;
text = text.replace(“foo”, “bar”);

// Met Pattern/Matcher wordt dit:
String text = “foo test foo”;
Matcher m = Pattern
      .compile(“foo”)
      .matcher(text);
text = m.replaceAll(“bar”);

// De RegEx vervanging kan ook
// rechtstreeks op String:
String text = “foo test foo”;
text = text
      .replaceAll(“foo”, “bar”);
```

Dit biedt weinig voordelen over de standaard `String.replace()` methode. Performance is vergelijkbaar. Bij veel vervangingen is `Pattern/Matcher` ongeveer 10 procent sneller. Een iets uitgebreider scenario wordt echter niet ondersteund door de standaard `String.replace()` methode. Dit is waar RegEx zijn kracht kan tonen.

```
// Vervang “foo” met “bar”, maar
// alleen als “foo” voorafgegaan
// wordt door een getal

String text = “123foo test foo”;
text = text.replaceAll(
      “(?<=[0-9])foo”,
      “bar”
);
```

Er wordt hier gebruik gemaakt van een positive look behind om te controleren of ‘foo’ voorafgegaan wordt door een nummer. Alleen als dat het geval is wordt ‘foo’ vervangen door ‘bar’. Een ander veel voorkomend gebruik van vervangen door middel van RegEx, is het invoegen van gevonden deelresultaten met een template. Neem het voorbeeld van de postcode. De expressie die we gebruiken matcht alle postcodes, ongeacht of ze een spatie bevatten. Als je alle postcodes in een tekst wilt opmaken zonder spaties, dan is dat een relatief moeilijke klus zonder reguliere expressies.

We passen de expressie voor de postcode iets aan: `\b([1-9][0-9]{3})[ ]([A-Z]{2})\b`.

Zoals je kunt zien zijn er twee capturing groups toegevoegd. Tevens is de spatie niet langer optioneel, we

willen immers alleen de postcodes aanpassen die nog niet voldoen aan ons gewenste formaat. Vervolgens schrijven we een template waar we ons resultaat in willen plaatsen: `$1$2`. Met één methodeaanroep kunnen we nu alle spaties uit de postcodes in een tekst verwijderen:

```
String text = "1212 MZ";

String pattern = "\\b([1-9][0-9]{3})[ ]([A-
Z]{2})\\b";
String template = "$1$2";
String result = text.replaceAll(
      pattern,
      template);

// Nu met Pattern/Matcher
Pattern p = Pattern
  .compile(pattern);
Matcher m = p.matcher(text);
result = m.replaceAll(template);
```

Elke capture group krijgt een nummer toegekend. 0 is de complete match, 1 is de eerste group en zo verder. Op deze manier kunnen maximaal 9 capture groupen worden geadresseerd. Om het teken $ op te kunnen nemen in de uitvoer moet deze worden ge-escapet met een backslash, dus: "\$".

Samenvattend:

| | |
|---|---|
| $0 | De gehele match. |
| $1 | De eerste capture group. |
| $2 | De tweede capture group, etc. |
| $9 | Negende en laatste capture group. |
| \$ | Het dollar teken. |

- ***Let op**: aangezien de backslash ook weer ge-escapet moet worden als je deze opneemt in een String, ziet het dollarteken er dus zo uit:* `\\$`.

Een andere methode om de spatie weg te halen is om niet de tekst te selecteren die we willen bewaren, maar juist te selecteren wat we weg willen halen en dit te vervangen met een lege tekst.

```
String text = "1212 MZ";
String pattern = "(?<=\\b[1-9]"
  + "[0-9]{3})[ ](?=[A-Z]{2}\\b)";

String result = text
  .replaceAll(pattern, "");
```

Hier wordt dus met look-arounds de spatie geselecteerd die verwijderd moet worden (in plaats van alle spaties). De look-behind controleert de cijfers, de look-ahead de letters van de postcode. De laatste mogelijkheid, een zelf geprogrammeerd algoritme gebruiken, is de krachtigste. Hiermee is het mogelijk om gevonden resultaten door middel van je eigen programmacode te manipuleren.

## Uiteindelijk hadden we slechts vijf verschillende alternatieven nodig om het schrikkeljaar te controleren

In de vervanging kun je gebruik maken van templates of een functie schrijven waarin je vervolgens weer een nieuwe RegEx gebruikt! We tonen hiervan een voorbeeld, waarin alle postcodes in een tekst worden opgemaakt zonder spaties en met de letters uppercase. Met dit voorbeeld als leidraad kun je zelf experimenteren om je eigen zoek en vervang functionaliteit te schrijven:

```
// Voorbeeld Postcode Manipulatie
String formatPC(String input)
{
  Pattern p = Pattern.compile(
    "(?:i)\\b([1-9][0-9]{3})\\s*"
  + "([A-Z]{2})\\b");

  StringBuffer sb =
    new StringBuffer();

  Matcher m = p.matcher(input);

  while (m.find())
  {
    m.appendReplacement(sb,
      "$1"
    + m.group(2).toUpperCase());
  }

  matcher.appendTail(sb);

  return sb.toString();
}
```

Als je zelf aan de slag gaat met RegEx is het raadzaam om, zeker in het begin, eerst op internet te zoeken naar de RegEx die je nodig hebt. Mogelijk dat deze niet genoeg controleert. Maar vanuit die basis kun je zelf de RegEx verbeteren.

Bovendien heeft het niet veel zin om telkens het wiel zelf uit te vinden...

- *Het is belangrijk om een expressie die je van anderen kopieert, uitgebreid te testen om te controleren of hij precies doet wat jij wilt!*

## Referenties


JavaDoc
http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/package-summary.html
http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Character.UnicodeBlock.html

Alternatieve RegEx implementaties
http://sourceforge.net/projects/jregex
http://jakarta.apache.org/regexp/index.html

RegEx bibliotheken & tutorials
http://regexlib.com/Default.aspx
http://www.regular-expressions.info/

RegEx testers
http://www.eclipse-plugins.info/eclipse/ plugin_details.jsp?id=964
http://brosinski.com/regex/
http://www.regexbuddy.com/index.html



*Jesse Houwing (e-mail: jesse.houwing @sogeti.nl) is werkzaam als Microsoft .Net Senior Application engineer bij Sogeti Nederland BV. Hij is de maker van de RegEx Fundamentals cursus die binnen Sogeti gegeven wordt.*

*Peter van den Berkmortel (e-mail: peter.vanden.berkmortel@sogeti.nl) is werkzaam als Technisch Architect in het Java Delivery Center van Sogeti. Hij is onder andere Certified JBoss Developer.*